

This application is submitted in the name of inventors Peter C. Damron and Nicolai Kosche, assignors to Sun Microsystems, Inc.

## SPECIFICATION

5

### SYSTEM AND METHOD FOR SCHEDULING INSTRUCTIONS TO MAXIMIZE OUTSTANDING PREFETCHES AND LOADS

#### RELATED APPLICATIONS

10 The present patent application is related to U.S. patent application Ser. No. \_\_\_\_\_  
(Attorney Docket No. SUN-P5370, Filed on \_\_\_\_) entitled "SYSTEM AND  
METHOD FOR SCHEDULING MEMORY INSTRUCTIONS TO PROVIDE  
ADEQUATE PREFETCH", Ser. No. \_\_\_\_\_ (Attorney Docket No. SUN-P5371,  
Filed on \_\_\_\_) entitled "HEURISTIC FOR IDENTIFYING LOADS

15 GUARANTEED TO HIT IN PROCESSOR CACHE", and \_\_\_\_\_ (Attorney  
Docket No. SUN-P5392, Filed on \_\_\_\_) entitled "SYSTEM AND METHOD FOR  
INSERTION OF PREFETCH INSTRUCTIONS BY A COMPILER", those  
applications having been assigned to the same assignee and being incorporated  
herein by reference.

20

## BACKGROUND OF THE INVENTION

### 1. Field of the Invention

5

This invention pertains generally to software compilers. More particularly this invention is directed to a system and method for the scheduling of prefetch and memory loading instructions by a compiler during compilation of software programs to maximize the efficiency of the code.

### 2. The Prior Art

10  
15  
15 Current computer systems include, among other things, a memory system and a processing unit (or processor or central processing unit (CPU)). A memory system serves as a repository of information, while the CPU accesses information from the memory system, operates on it, and stores it back.

20 It is well known that CPU clock speeds are increasing at a faster rate than memory speeds. This creates a time gap, typically measured in clock cycles of the

CPU, between the request for information in memory to when the information is available inside the CPU. If a CPU is executing instructions in a linear manner, when a currently executing instruction needs to read a memory location from the memory system, the request is “very urgent”. The processor must wait, or stalls, 5 while the memory system provides the data requested to the CPU. The number of CPU clock cycles between the clock cycle when the memory request was made to the cycle where the data is available to the instruction that needed it in the CPU is called the latency of the memory.

10 Caches are used to help alleviate the latency problem when reading from main memory. A cache is specially configured, high-speed, expensive memory in addition to the conventional memory (or main memory). FIG. 1 depicts a conventional hierarchical memory system, where a CPU is operatively coupled to a cache, and the cache is operatively coupled to the main memory. By placing the cache (small, relatively fast, expensive memory) between main memory (large, relatively slow memory) and the CPU, the memory system as a whole system is able to satisfy a substantial number of requests from the CPU at the speed of the cache, thereby reducing the overall latency of the system.

15 20 When the data requested by the CPU is in the cache (known as a “hit”), the request is satisfied at the speed of the cache. However, when the data requested by the CPU is not in the cache (known as a “miss”), the CPU must wait until the

data is provided from the slower main memory to the cache and then to the CPU, resulting in greater latency. As is well known in the art, the frequency of cache misses is much higher in some applications when compared to others. In particular, commercial systems employing databases (as most servers do) miss cache with much greater frequency than many systems running scientific or engineering applications.

To help address the problem of latency and to increase the hit to miss ratio associated with cache memory, many computer systems have included instructions for prefetching data from memory to cache. For example, instructions set architectures (ISA's), such as SPARC™ V9, support software data prefetch operations. The instruction's use, however, is left entirely to the program executing in the CPU. It may not be used at all, or it may be used with little or no intelligence, adding little in the way added performance. Because the level of knowledge needed about the CPU and its memory is extremely detailed in order to effectively use prefetch instructions, their use is generally left to compilers. For compilers to effectively use prefetch instructions, effective algorithms are needed which can be implemented by the compiler writers.

20 The algorithms needed for scientific and engineering applications is not as complex as for many commercial systems. This is due to the fact that scientific and engineering applications tend to work on arrays that generally reside in

contiguous memory locations. Thus, predicting which memory addresses will be required for the executing instruction stream is both relatively easy to predict and can be predicted in time to address latency concerns. In applications with regular memory reference patterns there will plenty of time to allow for the latency

5 between the issuing of the memory prefetch instruction, and the time when an executing instruction needs the contents of that memory location.

For database applications and other commercial applications, however, predicting which areas of memory will be required is much more difficult.

10 Because of the nature of the programs, there can be and often is a need for the contents of memory locations that are not contiguous. In addition to the non-contiguous memory locations, the compiled programs rarely leave enough time between instructions that will create a cache-miss and any associated prefetch instruction which may have alleviated the cache-miss otherwise. This means that 15 there is often insufficient latency time (in CPU cycles) between the address forming operation and the memory operation (associated with the address) to cover the prefetch latency. In these cases, there is no readily discernable way of establishing when a prefetch instruction should be issued to minimize latency.

20 Accordingly, there is a need for a method and apparatus which can schedule memory prefetch instructions such that the number of times adequate latency for

the prefetch instruction is provided can be maximized. The present invention satisfies this need and other deficiencies found in the background art.

#### BRIEF DESCRIPTION OF THE INVENTION

5

In an optimizing compiler, a new method and device for scheduling memory operations is disclosed. The present invention minimizes processor stalls due to the latency of loads or prefetches by scheduling memory operations to keep the target processors' memory fetch queue as full as possible. For applications 10 that experience large numbers of cache misses, scheduling for memory queues has proven to be more important than scheduling for the CPU pipeline itself.

The compiler includes a module for creating a directed acyclic graph (DAG) showing the dependencies of operations found in the code being compiled. 15 The compiler has a module that traverses the graph. Traversing the graph identifies nodes without dependencies on previous operations, which correspond to nodes having no incoming dependency edges. A node with no incoming dependencies corresponds to an instruction that is ready to execute. The group of nodes having no incoming dependency lines originating from other nodes form a 20 set of nodes called ready nodes. Any one of this set could be scheduled to run

next if no consideration is given to the need to keep the processors' in-flight memory queue as full as possible.

The set of ready nodes is compared to a model of the contents of a memory queue (or queues) kept by the compiler. If the queue has less than a designated optimal number of elements or entries, then a node corresponding to an instruction that adds to the queue is chosen. If the queue is already at the maximal number, then a memory operation that will remove an element from the queue is chosen.

Generally, loads, prefetches and stores add to the queue while operations that require that a previous value obtained from memory be available empty the queue.

If there is no optimal choice, any memory operation is chosen.

Once the choice has been made, a set of operations occur. The graph is updated by removing the chosen node along with any dependency edges that originated from it. The chosen node no longer belongs to the ready set, and the corresponding instruction is inserted in the code stream. The queue is updated accordingly (the corresponding element is added or some elements removed).

Once the updates have finished, the graph traversal module traverses the new graph and identifies a new set of ready nodes. Again, the best choice between the ready nodes is made using the heuristic keeping the queue or queues full. This

process repeats until traversing the graph yields no ready nodes. This corresponds to having no more instructions to schedule, and the algorithm is finished.

#### BRIEF DESCRIPTION OF THE DRAWING FIGURES

5      Figure 1 is a block diagram of prior art CPU-cache-memory.  
Figure 2 is a block diagram of conventional compiler processing.  
Figure 3 is a set of diagrams showing directed acyclic graphs.  
Figure 4 is a block diagram of system running the present invention.  
Figure 5 is a block diagram of a compiler embodying the present invention.  
Figure 6 is a block diagram of a method for the present invention..

#### DETAILED DESCRIPTION OF PREFERRED EMBODIMENTS

15      Person of ordinary skill in the art will realize that the following description of the present invention is illustrative only and not in any way limiting. Other embodiments of the invention will readily suggest themselves to such skilled persons having the benefit of this disclosure.

FIG. 2 shows the stages involved in a typical compiler. The stages will be mentioned briefly here in order to provide context for the present invention. For more detailed descriptions of these steps any of the standard books on the design and construction of software (not silicon) compilers can be used to supplement this 5 description. An example of a standard book on the subject is "Compilers: Principles, Techniques, and Tools" by Aho, Sethi, and Ullman.

The input to a compiler is the program to be compiled. This may constitute anything from a part of a small single file containing a few instructions (or lines of 10 code) to a large number of files contain thousands and thousands of instructions (or lines of code). From the compiler's viewpoint, this is instruction stream 200.

The initial step is identify the basic operational components of the instruction stream, the tokens. This is done in the scanner, 202. The output of the 15 scanner is a token stream 204. All the elements of the token stream are recognized ("legal") tokens of the language being parsed. Token stream 204 is fed into parser 206. Parser 206 analyzes the syntax of the token stream. If the program was syntactically correct, parser 206 creates as output a parse tree 208, which contains all the tokens in token stream 204. The parse tree 208 is used by intermediate 20 representation generator 210 to generate the program in an high-level intermediate representation 212. The intermediate representation is then passed to intermediate representation optimizer 214. Intermediate representation optimizer 214 optimizes

the program in ways suitable for this high level, such as doing a global intraprocedural analysis of the program's control and data flows. As is well known in the art, optimizations are carried out using more than one type of optimizing algorithm. Typically the compiler designers will include as many as 5 they think they need to achieve their design goals.

The output from intermediate representation optimizer 214 is an optimized high level intermediate representation of the program, 216. This is then given to low level intermediate representation generator 218. Low level intermediate representation generator 218 generates lines of code 220 that are very close to, or are, assembly language statements for the target processor. This code stream 220 is then fed to low level intermediate optimizer 222. Low level intermediate optimizer 222 carries out optimizations appropriate for that level of code, such as software pipelining optimizations, procedure optimizations, register optimizations, and instruction scheduling. As with the high level optimizers, the compiler 10 designers will include as many optimizing steps and algorithms as they think they 15 need to achieve their design goals. The output, 224, is an optimized version of low level intermediate representation 220. This is fed to relocatable assembly code generator 226, which generates relocatable assembly code located in one of 20 more files, 228.

Some compilers and compiler designers don't use two levels of intermediate representations. Although it can be argued the difference is more semantic than content-based, in such cases 218 generates a version of assembly code, 222 optimizes the assembly code, and 226 creates relocatable assembly code 5 or relocatable object code files as output. The invention in the present disclosure will work equally well in either case.

The present invention is a new optimization method and system for optimizing in the low level intermediate representation, or assembly code level, 10 optimizer, 222. It is intended to be embodied in a compiler for use with any processor or processor family that may benefit from the disclosed optimization. One such family of processors is the UltraSPARC family, in particular the UltraSPARC III, from Sun Microsystems, Inc.

15 As is well known in the art dependency graphs, and in particular directed acyclic graphs (DAGs), are commonly used to map or graph dependencies between instructions. Dependency graphs have been used in the past to help optimize instruction scheduling for processor pipelines, or processors with multiple execution units.

20

The present invention came about through experimental confirmation that graphs preserving dependencies, and in particular DAGs, could be successfully

used as part of a solution to optimize instruction scheduling for memory operation instructions or memory operations. This is a new use, application, and method

involving dependency graphs such as DAGs. For any section of code at the assembly (or low-level intermediate) level, a DAG may be created showing the

5 dependencies between memory operations. An example is given below. The example is in pseudo-code, where registers are indicated by the prefix "r" followed by a number – r6 is register 6, square brackets indicate the "contents at address" – [r3] means the contents when r3 is used as an address, a non-memop has three arguments including the two operands and a target register – "add r1, r2, r3" means add r1 and r2 and place the result in r3, and the line numbers are indicated by "Ln" where n is an integer.

10  
15  
20  
25  
30  
35  
40  
45  
50  
55  
60  
65  
70  
75  
80  
85  
90  
95  
100  
105  
110  
115  
120  
125  
130  
135  
140  
145  
150  
155  
160  
165  
170  
175  
180  
185  
190  
195  
200  
205  
210  
215  
220  
225  
230  
235  
240  
245  
250  
255  
260  
265  
270  
275  
280  
285  
290  
295  
300  
305  
310  
315  
320  
325  
330  
335  
340  
345  
350  
355  
360  
365  
370  
375  
380  
385  
390  
395  
400  
405  
410  
415  
420  
425  
430  
435  
440  
445  
450  
455  
460  
465  
470  
475  
480  
485  
490  
495  
500  
505  
510  
515  
520  
525  
530  
535  
540  
545  
550  
555  
560  
565  
570  
575  
580  
585  
590  
595  
600  
605  
610  
615  
620  
625  
630  
635  
640  
645  
650  
655  
660  
665  
670  
675  
680  
685  
690  
695  
700  
705  
710  
715  
720  
725  
730  
735  
740  
745  
750  
755  
760  
765  
770  
775  
780  
785  
790  
795  
800  
805  
810  
815  
820  
825  
830  
835  
840  
845  
850  
855  
860  
865  
870  
875  
880  
885  
890  
895  
900  
905  
910  
915  
920  
925  
930  
935  
940  
945  
950  
955  
960  
965  
970  
975  
980  
985  
990  
995  
1000  
1005  
1010  
1015  
1020  
1025  
1030  
1035  
1040  
1045  
1050  
1055  
1060  
1065  
1070  
1075  
1080  
1085  
1090  
1095  
1100  
1105  
1110  
1115  
1120  
1125  
1130  
1135  
1140  
1145  
1150  
1155  
1160  
1165  
1170  
1175  
1180  
1185  
1190  
1195  
1200  
1205  
1210  
1215  
1220  
1225  
1230  
1235  
1240  
1245  
1250  
1255  
1260  
1265  
1270  
1275  
1280  
1285  
1290  
1295  
1300  
1305  
1310  
1315  
1320  
1325  
1330  
1335  
1340  
1345  
1350  
1355  
1360  
1365  
1370  
1375  
1380  
1385  
1390  
1395  
1400  
1405  
1410  
1415  
1420  
1425  
1430  
1435  
1440  
1445  
1450  
1455  
1460  
1465  
1470  
1475  
1480  
1485  
1490  
1495  
1500  
1505  
1510  
1515  
1520  
1525  
1530  
1535  
1540  
1545  
1550  
1555  
1560  
1565  
1570  
1575  
1580  
1585  
1590  
1595  
1600  
1605  
1610  
1615  
1620  
1625  
1630  
1635  
1640  
1645  
1650  
1655  
1660  
1665  
1670  
1675  
1680  
1685  
1690  
1695  
1700  
1705  
1710  
1715  
1720  
1725  
1730  
1735  
1740  
1745  
1750  
1755  
1760  
1765  
1770  
1775  
1780  
1785  
1790  
1795  
1800  
1805  
1810  
1815  
1820  
1825  
1830  
1835  
1840  
1845  
1850  
1855  
1860  
1865  
1870  
1875  
1880  
1885  
1890  
1895  
1900  
1905  
1910  
1915  
1920  
1925  
1930  
1935  
1940  
1945  
1950  
1955  
1960  
1965  
1970  
1975  
1980  
1985  
1990  
1995  
2000  
2005  
2010  
2015  
2020  
2025  
2030  
2035  
2040  
2045  
2050  
2055  
2060  
2065  
2070  
2075  
2080  
2085  
2090  
2095  
2100  
2105  
2110  
2115  
2120  
2125  
2130  
2135  
2140  
2145  
2150  
2155  
2160  
2165  
2170  
2175  
2180  
2185  
2190  
2195  
2200  
2205  
2210  
2215  
2220  
2225  
2230  
2235  
2240  
2245  
2250  
2255  
2260  
2265  
2270  
2275  
2280  
2285  
2290  
2295  
2300  
2305  
2310  
2315  
2320  
2325  
2330  
2335  
2340  
2345  
2350  
2355  
2360  
2365  
2370  
2375  
2380  
2385  
2390  
2395  
2400  
2405  
2410  
2415  
2420  
2425  
2430  
2435  
2440  
2445  
2450  
2455  
2460  
2465  
2470  
2475  
2480  
2485  
2490  
2495  
2500  
2505  
2510  
2515  
2520  
2525  
2530  
2535  
2540  
2545  
2550  
2555  
2560  
2565  
2570  
2575  
2580  
2585  
2590  
2595  
2600  
2605  
2610  
2615  
2620  
2625  
2630  
2635  
2640  
2645  
2650  
2655  
2660  
2665  
2670  
2675  
2680  
2685  
2690  
2695  
2700  
2705  
2710  
2715  
2720  
2725  
2730  
2735  
2740  
2745  
2750  
2755  
2760  
2765  
2770  
2775  
2780  
2785  
2790  
2795  
2800  
2805  
2810  
2815  
2820  
2825  
2830  
2835  
2840  
2845  
2850  
2855  
2860  
2865  
2870  
2875  
2880  
2885  
2890  
2895  
2900  
2905  
2910  
2915  
2920  
2925  
2930  
2935  
2940  
2945  
2950  
2955  
2960  
2965  
2970  
2975  
2980  
2985  
2990  
2995  
3000  
3005  
3010  
3015  
3020  
3025  
3030  
3035  
3040  
3045  
3050  
3055  
3060  
3065  
3070  
3075  
3080  
3085  
3090  
3095  
3100  
3105  
3110  
3115  
3120  
3125  
3130  
3135  
3140  
3145  
3150  
3155  
3160  
3165  
3170  
3175  
3180  
3185  
3190  
3195  
3200  
3205  
3210  
3215  
3220  
3225  
3230  
3235  
3240  
3245  
3250  
3255  
3260  
3265  
3270  
3275  
3280  
3285  
3290  
3295  
3300  
3305  
3310  
3315  
3320  
3325  
3330  
3335  
3340  
3345  
3350  
3355  
3360  
3365  
3370  
3375  
3380  
3385  
3390  
3395  
3400  
3405  
3410  
3415  
3420  
3425  
3430  
3435  
3440  
3445  
3450  
3455  
3460  
3465  
3470  
3475  
3480  
3485  
3490  
3495  
3500  
3505  
3510  
3515  
3520  
3525  
3530  
3535  
3540  
3545  
3550  
3555  
3560  
3565  
3570  
3575  
3580  
3585  
3590  
3595  
3600  
3605  
3610  
3615  
3620  
3625  
3630  
3635  
3640  
3645  
3650  
3655  
3660  
3665  
3670  
3675  
3680  
3685  
3690  
3695  
3700  
3705  
3710  
3715  
3720  
3725  
3730  
3735  
3740  
3745  
3750  
3755  
3760  
3765  
3770  
3775  
3780  
3785  
3790  
3795  
3800  
3805  
3810  
3815  
3820  
3825  
3830  
3835  
3840  
3845  
3850  
3855  
3860  
3865  
3870  
3875  
3880  
3885  
3890  
3895  
3900  
3905  
3910  
3915  
3920  
3925  
3930  
3935  
3940  
3945  
3950  
3955  
3960  
3965  
3970  
3975  
3980  
3985  
3990  
3995  
4000  
4005  
4010  
4015  
4020  
4025  
4030  
4035  
4040  
4045  
4050  
4055  
4060  
4065  
4070  
4075  
4080  
4085  
4090  
4095  
4100  
4105  
4110  
4115  
4120  
4125  
4130  
4135  
4140  
4145  
4150  
4155  
4160  
4165  
4170  
4175  
4180  
4185  
4190  
4195  
4200  
4205  
4210  
4215  
4220  
4225  
4230  
4235  
4240  
4245  
4250  
4255  
4260  
4265  
4270  
4275  
4280  
4285  
4290  
4295  
4300  
4305  
4310  
4315  
4320  
4325  
4330  
4335  
4340  
4345  
4350  
4355  
4360  
4365  
4370  
4375  
4380  
4385  
4390  
4395  
4400  
4405  
4410  
4415  
4420  
4425  
4430  
4435  
4440  
4445  
4450  
4455  
4460  
4465  
4470  
4475  
4480  
4485  
4490  
4495  
4500  
4505  
4510  
4515  
4520  
4525  
4530  
4535  
4540  
4545  
4550  
4555  
4560  
4565  
4570  
4575  
4580  
4585  
4590  
4595  
4600  
4605  
4610  
4615  
4620  
4625  
4630  
4635  
4640  
4645  
4650  
4655  
4660  
4665  
4670  
4675  
4680  
4685  
4690  
4695  
4700  
4705  
4710  
4715  
4720  
4725  
4730  
4735  
4740  
4745  
4750  
4755  
4760  
4765  
4770  
4775  
4780  
4785  
4790  
4795  
4800  
4805  
4810  
4815  
4820  
4825  
4830  
4835  
4840  
4845  
4850  
4855  
4860  
4865  
4870  
4875  
4880  
4885  
4890  
4895  
4900  
4905  
4910  
4915  
4920  
4925  
4930  
4935  
4940  
4945  
4950  
4955  
4960  
4965  
4970  
4975  
4980  
4985  
4990  
4995  
5000  
5005  
5010  
5015  
5020  
5025  
5030  
5035  
5040  
5045  
5050  
5055  
5060  
5065  
5070  
5075  
5080  
5085  
5090  
5095  
5100  
5105  
5110  
5115  
5120  
5125  
5130  
5135  
5140  
5145  
5150  
5155  
5160  
5165  
5170  
5175  
5180  
5185  
5190  
5195  
5200  
5205  
5210  
5215  
5220  
5225  
5230  
5235  
5240  
5245  
5250  
5255  
5260  
5265  
5270  
5275  
5280  
5285  
5290  
5295  
5300  
5305  
5310  
5315  
5320  
5325  
5330  
5335  
5340  
5345  
5350  
5355  
5360  
5365  
5370  
5375  
5380  
5385  
5390  
5395  
5400  
5405  
5410  
5415  
5420  
5425  
5430  
5435  
5440  
5445  
5450  
5455  
5460  
5465  
5470  
5475  
5480  
5485  
5490  
5495  
5500  
5505  
5510  
5515  
5520  
5525  
5530  
5535  
5540  
5545  
5550  
5555  
5560  
5565  
5570  
5575  
5580  
5585  
5590  
5595  
5600  
5605  
5610  
5615  
5620  
5625  
5630  
5635  
5640  
5645  
5650  
5655  
5660  
5665  
5670  
5675  
5680  
5685  
5690  
5695  
5700  
5705  
5710  
5715  
5720  
5725  
5730  
5735  
5740  
5745  
5750  
5755  
5760  
5765  
5770  
5775  
5780  
5785  
5790  
5795  
5800  
5805  
5810  
5815  
5820  
5825  
5830  
5835  
5840  
5845  
5850  
5855  
5860  
5865  
5870  
5875  
5880  
5885  
5890  
5895  
5900  
5905  
5910  
5915  
5920  
5925  
5930  
5935  
5940  
5945  
5950  
5955  
5960  
5965  
5970  
5975  
5980  
5985  
5990  
5995  
6000  
6005  
6010  
6015  
6020  
6025  
6030  
6035  
6040  
6045  
6050  
6055  
6060  
6065  
6070  
6075  
6080  
6085  
6090  
6095  
6100  
6105  
6110  
6115  
6120  
6125  
6130  
6135  
6140  
6145  
6150  
6155  
6160  
6165  
6170  
6175  
6180  
6185  
6190  
6195  
6200  
6205  
6210  
6215  
6220  
6225  
6230  
6235  
6240  
6245  
6250  
6255  
6260  
6265  
6270  
6275  
6280  
6285  
6290  
6295  
6300  
6305  
6310  
6315  
6320  
6325  
6330  
6335  
6340  
6345  
6350  
6355  
6360  
6365  
6370  
6375  
6380  
6385  
6390  
6395  
6400  
6405  
6410  
6415  
6420  
6425  
6430  
6435  
6440  
6445  
6450  
6455  
6460  
6465  
6470  
6475  
6480  
6485  
6490  
6495  
6500  
6505  
6510  
6515  
6520  
6525  
6530  
6535  
6540  
6545  
6550  
6555  
6560  
6565  
6570  
6575  
6580  
6585  
6590  
6595  
6600  
6605  
6610  
6615  
6620  
6625  
6630  
6635  
6640  
6645  
6650  
6655  
6660  
6665  
6670  
6675  
6680  
6685  
6690  
6695  
6700  
6705  
6710  
6715  
6720  
6725  
6730  
6735  
6740  
6745  
6750  
6755  
6760  
6765  
6770  
6775  
6780  
6785  
6790  
6795  
6800  
6805  
6810  
6815  
6820  
6825  
6830  
6835  
6840  
6845  
6850  
6855  
6860  
6865  
6870  
6875  
6880  
6885  
6890  
6895  
6900  
6905  
6910  
6915  
6920  
6925  
6930  
6935  
6940  
6945  
6950  
6955  
6960  
6965  
6970  
6975  
6980  
6985  
6990  
6995  
7000  
7005  
7010  
7015  
7020  
7025  
7030  
7035  
7040  
7045  
7050  
7055  
7060  
7065  
7070  
7075  
7080  
7085  
7090  
7095  
7100  
7105  
7110  
7115  
7120  
7125  
7130  
7135  
7140  
7145  
7150  
7155  
7160  
7165  
7170  
7175  
7180  
7185  
7190  
7195  
7200  
7205  
7210  
7215  
7220  
7225  
7230  
7235  
7240  
7245  
7250  
7255  
7260  
7265  
7270  
7275  
7280  
7285  
7290  
7295  
7300  
7305  
7310  
7315  
7320  
7325  
7330  
7335  
7340  
7345  
7350  
7355  
7360  
7365  
7370  
7375  
7380  
7385  
7390  
7395  
7400  
7405  
7410  
7415  
7420  
7425  
7430  
7435  
7440  
7445  
7450  
7455  
7460  
7465  
7470  
7475  
7480  
7485  
7490  
7495  
7500  
7505  
7510  
7515  
7520  
7525  
7530  
7535  
7540  
7545  
7550  
7555  
7560  
7565  
7570  
7575  
7580  
7585  
7590  
7595  
7600  
7605  
7610  
7615  
7620  
7625  
7630  
7635  
7640  
7645  
7650  
7655  
7660  
7665  
7670  
7675  
7680  
7685  
7690  
7695  
7700  
7705  
7710  
7715  
7720  
7725  
7730  
7735  
7740  
7745  
7750  
7755  
7760  
7765  
7770  
7775  
7780  
7785  
7790  
7795  
7800  
7805  
7810  
7815  
7820  
7825  
7830  
7835  
7840  
7845  
7850  
7855  
7860  
7865  
7870  
7875  
7880  
7885  
7890  
7895  
7900  
7905  
7910  
7915  
7920  
7925  
7930  
7935  
7940  
7945  
7950  
7955  
7960  
7965  
7970  
7975  
7980  
7985  
7990  
7995  
8000  
8005  
8010  
8015  
8020  
8025  
8030  
8035  
8040  
8045  
8050  
8055  
8060  
8065  
8070  
8075  
8080  
8085  
8090  
8095  
8100  
8105  
8110  
8115  
8120  
8125  
8130  
8135  
8140  
8145  
8150  
8155  
8160  
8165  
8170  
8175  
8180  
8185  
8190  
8195  
8200  
8205  
8210  
8215  
8220  
8225  
8230  
8235  
8240  
8245  
8250  
8255  
8260  
8265  
8270  
8275  
8280  
8285  
8290  
8295  
8300  
8305  
8310  
8315  
8320  
8325  
8330  
8335  
8340  
8345  
8350  
8355  
8360  
8365  
8370  
8375  
8380  
8385  
8390  
8395  
8400  
8405  
8410  
8415  
8420  
8425  
8430  
8435  
8440  
8445  
8450  
8455  
8460  
8465  
8470  
8475  
8480  
8485  
8490  
8495  
8500  
8505  
8510  
8515  
8520  
8525  
8530  
8535  
8540  
8545  
8550  
8555  
8560  
8565  
8570  
8575  
8580  
8585  
8590  
8595  
8600  
8605  
8610  
8615  
8620  
8625  
8630  
8635  
8640  
8645  
8650  
8655  
8660  
8665  
8670  
8675  
8680  
8685  
8690  
8695  
8700  
8705  
8710  
8715  
8720  
8725  
8730  
8735  
8740  
8745  
8750  
8755  
8760  
8765  
8770  
8775  
8780  
8785  
8790  
8795  
8800  
8805  
8810  
8815  
8820  
8825  
8830  
8835  
8840  
8845  
8850  
8855  
8860  
8865  
8870  
8875  
8880  
8885  
8890  
8895  
8900  
8905  
8910  
8915  
8920  
8925  
8930  
8935  
8940  
8945  
8950  
8955  
8960  
8965  
8970  
8975  
8980  
8985  
8990  
8995  
9000  
9005  
9010  
9015  
9020  
9025  
9030  
9035  
9040  
9045  
9050  
9055  
9060  
9065  
9070  
9075  
9080  
9085  
9090  
9095  
9100  
9105  
9110  
9115  
9120  
9125  
9130  
9135  
9140  
9145  
9150  
9155  
9160  
9165  
9170  
9175  
9180  
9185  
9190  
9195  
9200  
9205  
9210  
9215  
9220  
9225  
9230  
9235  
9240  
9245  
9250  
9255  
9260  
9265  
9270  
9275  
9280  
9285  
9290  
9295  
9300  
9305  
9310  
9315  
9320  
9325  
9330  
9335  
9340  
9345  
9350  
9355  
9360  
9365  
9370  
9375  
9380  
9385  
93

DAG is shown in 3A. Any node with an incoming edge is dependent on another instruction; only those nodes with no incoming edges are ready to execute. In this case that means nodes L1 and L3 are equally ready to execute.

5        Looking at lines L1 and L3, it is readily seen this is true. Either could be executed first. A completely arbitrary choice is made to execute L1 first. As soon as that choice is made, the node with L1 in it is removed from the graph, as well as any dependency edges originating from it (after L1 executes there will no longer be anything depending on it to finish – the results are available). The 10      resulting graph is shown in FIG. 3B. The instructions that are ready to execute in the new graph correspond to L2, L3 and L4. A quick check of the actual lines of code will show this is correct. Again making an arbitrary choice between the two, L3 will be scheduled.

15        After removing L3 and its outbound dependency edges, the resulting graph is shown in FIG. 3C. Any one of the instructions corresponding to nodes L2, L4, or L6 may be scheduled. An arbitrary choice of scheduling node L2 first is made. The resulting graph, after removing L2 and any outbound dependency edges (there were none, in this case) is shown in FIG 3D. Now either of nodes L4 or L6 may 20      be chosen to be scheduled. L6 is chosen and removed, leaving the graph shown in FIG 3E. The only choice left is L4, which is scheduled and removed, leaving the

graph shown in FIG 3F. The final remaining node, L5, is scheduled next. After removing the node L5, the graph is empty.

The use of the DAG helps in making sure that the instructions are scheduled such that no instructions having dependencies are incorrectly, but what about the situations when there were multiple choices to be made – multiple nodes all with no dependencies and available at the same time? In the above example purely arbitrary choices between available nodes were made.

10 It has been experimentally determined that a significant boost in processor performance can be realized by adding another step to the use of DAGs for memory operations. Rather than arbitrarily choosing any of the available ready-to-execute nodes from the DAG, the heuristic use of a memory queue is added. The memory queue discipline is that of a FIFO queue. Any time there is a choice 15 of memory operations in the DAG, a memory operation is chosen that will tend to fill the memory queue. This continues until the memory queue has a predefined number of elements, or is full. Given a choice, memory operations are now chosen that will keep the memory queue as nearly full as possible without overfilling.

20

In target processors, this corresponds to ordering instructions so that the maximum number of outstanding loads, prefetches, or stores are always in the

processor's outstanding memory operation queue. This helps maximize the parallelism in the memory system, and minimize or eliminate the processor getting into a stall state. The queue will be sized to model a memory operations queue on a target processor, for example the UltraSPARC III has an outstanding memory operation queue length of 8. Having the memory operation queue as full as possible helps to get more memory operations started and running, and thus will maximize cache hits and prevent processor stall.

10 The actual processor may or may not stall if the queue is filled, depending on the status of the earliest entry. Since the goal is keep the processor from stalling, the target for filling the actual hardware queue will be to keep the queue as close to one less than the maximum number of entries allowed as possible. This same goal is then used in the software queue of the optimizing compiler.

15 While using memory queue(s) in an optimizing compiler, memory operations are categorized as either adding or subtracting elements from the memory operations queue (or queues – for ease of discussion, one queue will be assumed). Generally loads and prefetches will add a memory operation element to the queue, while operations that rely on a memory operation to complete in order 20 to execute will remove an element or elements from the queue. Stores are more difficult to model because in most cases there is no specific instruction to cause the store to complete and be removed from the memory queue.

It will be appreciated by a person of average skill in this art and with the benefit of the present disclosure that other variations of the heuristic just discussed can either be added or subtracted from the algorithms used in determining how 5 individual instructions should be scheduled. All such variations, as well as other heuristics and choice methods, are anticipated by the present disclosure and are within the inventive nature of using DAGs and queues for optimizing the order of memory operation instructions for a target processor.

10 Referring now to FIG 4, a block diagram of a system running the present invention is shown. Computing device 400 has a hardware base 402 and an operating system 404. These two components are of conventional design using well known components and are exemplified by a Sun Microsystems Ultra 60 Workstation running the Solaris 7 operating system. The hardware and operating 15 system must be minimally adaptable and configurable to act as support to run the present invention, such systems typically going under than name of workstations or servers. Any system providing enough system resources will work with the present invention. On top of operating system 404 are various applications, shown as 406, 408, 410, and 412. The arrows indicated by 414 show the parts of the 20 system that are directly interacting with each other. Operating system 404 is continually interacting with hardware 402 as well as each of the applications 406, 308, 410 and 412. The application programs 406, 408, 410, and 412 make use of

system services, system resources, and hardware resources through operating system 404.

Application 412 is a compiler program according to the present invention.

5 The application programs 406, 408, and 410 are shown for illustrative purposes only, and may or may not exist in any particular system on which the present invention is installed. They are not required nor are they a part of the present invention; they exist independently of, and may coexist with, the present invention.

10 Overall, compiler application 412 contains many functional modules including those needed to compile any given source code program into relocatable assembly code. The implementation of a compiler with modules according to the present invention may be implemented in a traditional programming language such as C or C++, or other suitable language. As will be appreciated by one of ordinary skill in the art and with the benefit of the present disclosure, the compiler may be implemented in any number of ways and with considerable variance in overall functionality while still encompassing the inventive nature of the present disclosure.

15

20

Continuing on in FIG. 4, compiler program 412 is reading from source files, using interim files, and writing to target files, shown as files 418, 420, and

422. The interactions between compiler program 412 and the files is indicated by communication arrows 428. Other source files, which may be library files or other system resident files, are shown as 424 and 426 and have not yet been called by the compiler. As will be appreciated by those of ordinary skill in the art and with 5 the benefit of the present invention, there may be any number of source, target, and interim files.

The server's main memory is shown as 440. 442 is memory space taken up by the operating system. 444 is the main memory space occupied by the compiler application program according to the present invention. Source files, target files, and interim files are shown as 446, 448, 450, and 452.

10 FIG. 5 is a logical block diagram of a compiler program according to the present invention. Compiler 500 reads from source files, shown as 502 and 504. 15 There may be any number of source files, indicated by the ellipses between source file 502 and source file 504. During the compilation process, various interim files are created, shown as files 518 and 520. There can be any number of such files, as shown by the ellipses between files 518 and 520. The overall process is controlled by process manager module 506.

20

When discussing functional module components implemented in software of the present invention, "module" is used to mean any collection of lines of code

found on machine readable media that perform the indicated function. Typically such lines of functionally related code will be organized into a closely associated set of routines, files, functions, and other software engineering elements.

However, regardless of how the actual lines of code are physically distributed in

5 the code base that provides the functionality being described, those lines of code comprise the embodiment of the functional blocks. Additionally, some modules may also be implemented in hardware. All such implementations are fully within the inventive concepts of the present disclosure.

10 After the source files are compiled into assembly code or a similar low level intermediate representation by other components of the compiler (shown generally as functional block 522), the assembly code files are passed by the process manager 506 to DAG creation module 508. While it is expected that process manager module 506 will be used in most implementations, the present invention also fully contemplates implementations without process manager 506.

15 Process manager 506 is an optional module.

After DAG creation module 508 completes the dependency graph, the graph is traversed by DAG traversal and ready set identification module 510.

20 DAGs are traversed in ways well known in the art; in this case, the traversal will be in topological order. Another property of the just created DAG is there will always be at least one node with no dependency arrows entering it. This is true

because there will always be the first instruction of the program which, because it is the first, cannot be dependent on any previous instruction. When there are no more nodes in the DAG there will be no more instructions to schedule, so the algorithm ends.

5

At any given time in the traversal of the DAG there will ordinarily be several, if not many, nodes that have no dependency arrows coming into them.

Any of these nodes may be inserted next in the instruction stream (they are all ready to be scheduled or to execute). This group of ready to execute nodes, or

10 instructions, are called the ready set. The ready set always starts with at least one member, and when it becomes empty (meaning there are no more nodes that can be scheduled), the algorithm is over. Nodes are subtracted from the ready set when a node element is chosen to be executed (scheduled, put next in the instruction stream by the optimizer). Nodes are added to the ready set after the

15 DAG traversal module traverses the new graph that is left after a node is removed.

When a node is removed, so are any edges originating from it. The DAG traversal module then re-evaluates the remaining graph, detecting any nodes elements that are now ready to be scheduled because of the removal of dependency arrows. The DAG traversal and ready set management duties are all carried out within DAG

20 traversal and ready set identification module 510.

Once the ready set has been updated or established, ready set evaluation module 512 has the task of evaluating the current state of the memory queue with the current members of the ready set. This is explained in more detail below, but briefly, ready set evaluation module 512 checks the memory queue for the number 5 of slots available. Module 512 then picks the best member of the ready set to keep the memory queue as full as possible without stalling the processor.

Queue management module 514 manages the memory queue (or queues).

This module adds members, removes members, and responds to requests about 10 current members from other modules.

The results of picking one member of the ready set over another as the DAG graph is traversed is implemented in the target compiled code by code 15 scheduling module 516. When a member of an ready set is chosen by ready set evaluation module 512, the corresponding instruction line is arranged to be the next executed in the code stream by code scheduling module 516.

The result of the overall compilation process is a relocatable assembly language file or files, shown as files 524 and 526. There may be any number of 20 output files, depending on the input to compiler 500, as is indicated by the ellipses between files 524 and 526.

Referring now to FIG 6, a method of using the present invention is shown in block diagram form. For each file or set of files that are compiled, the first step is the creation of a directed acyclic graph that represents the dependencies between instructions (DAG). This action is shown in block 600. After the DAG graph creation step is completed, the process continues to block 602.

There are two related actions carried out in block 602. The DAG is traversed in topological order, visiting the nodes as specified by the ordering. The traversal order and ready or executable node identification methods are those which are well-known in the art; see generally "Introduction To Automata Theory, Languages, And Computation" by Hopcroft and Ullman, as well as the compiler text referenced earlier. Once the graph has been updated, traversed, and the current ready set identified, block 602 is left and diamond 604 entered.

Diamond 604 queries if the ready set is empty. If it is, the "YES" exit is taken to oval 606, the end of the process. If oval 606 is reached all the instruction scheduling for the file or files being compiled has been completed, and this iteration of instruction scheduling is complete. If the ready set is not empty, the "NO" exit is taken to diamond 608.

20

Diamond 608 evaluates the state of the memory queue. If the memory queue has fewer elements in it than one less than the number which corresponds to

execution being blocked in the target processor (the total number of slots, the queue's size, less one), then the "YES" exit is taken to diamond 610.

Diamond 610 determines if there is a memory operation in the ready set  
5 that will add an element to the queue, called a mem-add node. The goal is to keep  
the number of elements in the memory operations queue as close to (queue\_size -  
1) as possible, where queue\_size is the size at which pipeline stalls of the actual  
in-flight memory fetch queue of the target processor will occur. If there is a node  
of the ready set that can be added to the memory queue, the "YES" exit is taken  
10 from diamond 610 and block 612 is entered.

The actions in block 612 involve first removing the chosen node from the  
ready set, adding the corresponding element to the memory queue (FIFO  
ordering), scheduling the instruction corresponding to the chosen element in the  
15 instruction stream, and removing the chosen node from the DAG (this includes  
removing any dependency edges originating from that node in the graph).

Looking a bit further at queuing and dequeuing in general, once an element  
corresponding to a mem-add node has been added to the memory queue, it will be  
20 dequeued when the first of its successor nodes is scheduled, and this node is called  
a mem-sub node. For example in figure 3A, when node L1 is scheduled an  
element is added to the memory queue, then the first of nodes L2 or L4 to be

scheduled will be the mem-sub node and remove that element plus any elements further back in the queue. So, at figure 3C, when L2 is scheduled, the queue element for L1 is removed, but L3 is not removed, since it was added to the queue (scheduled) after L1.

5

Returning to box 612, after these actions are completed block 612 is exited and block 602 entered. The process will now continue with block 602, carrying out the DAG traversal and identification of the current ready set.

10        If, at diamond 610 there was no memory operation in the ready set that can be added to the memory operations queue, then the remainder of the entries in the ready set must be memory operations that remove elements from the memory operations queue. The "NO" exit is taken from diamond 610, and block 618 is entered.

15        Since there are no memory-add node types available in the ready set, the action taken in block 618 is to choose a next best node. If there are multiple nodes in the ready set, then one may be chosen arbitrarily or other heuristics may be employed. If there are multiple memory queues, they may each be assigned a priority or a weight that can be used to help choose which node should be scheduled first (e.g. adding an element to the highest priority queue would be preferred). The simplest heuristic is to choose the first ready node or to choose an

arbitrary node, as in the example of figures 3A-3F. A common heuristic is to choose the highest node from the bottom of the graph. In actual practice, there are several heuristics used in sequence, where a tie in one heuristic leads to the use of the next heuristic, as is well known in the art.

5

After a node is chosen, the actions taken in block 618 involve first removing the chosen node from the ready set. If the node corresponds to a mem-sub instruction then one or more elements will be removed from a queue. If the chosen node corresponds to an instruction that is neither, no change is made to any memory queue.

If the node corresponded to a mem-sub instruction, the element in the memory queue furthest from the end of the memory queue, and corresponding to node just removed from the ready set (if there is one), is removed from the memory queue along with any elements closer to the end of the queue (if any), in keeping with FIFO ordering.

This corresponds to the idea that if a memory operation has completed for an element of the memory queue that is not the last element of the queue, any previously started memop operations will also have completed. The correspondence between nodes being scheduled and queue elements is that of load to use, a prefetch to memory operation, or load to store. This correspondence is

also represented by an edge in the DAG where the memory operation is a successor node. It is also the case that some operations in subroutine calls and cache misses may also drain some of the memory queue elements.

5 Next, the instruction corresponding to the chosen node is inserted in the instruction stream. Finally, the chosen node is removed from the DAG, including the removal of any dependency edges originating from that node.

10 After these actions are completed, block 618 is exited and block 602 entered. The process will now continue with block 602, carrying out the DAG traversal and identification of the current ready set.

15 Returning now to diamond 608, if the number of elements currently in the memory queue is greater than or equal to (queue\_size – 1), then the “NO” exit is taken to diamond 614. Note that it is entirely possible that the memory queue size may grow larger than (queue\_size). This undesirable condition would correspond to the target processor being in a stall state while outstanding memory operations are completed, and is what is being minimized and/or eliminated by the current invention.

20

Diamond 614 searches for an available node in the ready set that has the effect of removing elements from the memory queue, called a mem-sub node. If

there is a mem-sub node in the ready set, then the "YES" exit is taken to block 616.

In block 616 the following actions are carried out: remove the chosen node  
5 from the ready set; remove the element corresponding to the chosen node from the  
memory queue (where "corresponding to" means the predecessor node of the  
chosen node that was a mem-add for this chosen node), plus any older elements if  
any; insert the instruction corresponding to the chosen ready set node in the  
instruction stream; and, remove the node corresponding to the chosen element  
10 from the DAG, including the removal of any dependency edges originating from  
that node in the graph.

After these actions are completed, block 616 is exited and block 602  
entered. The process will now continue with block 602, carrying out the DAG  
15 traversal and identification of the current ready set.

Finally, if there are no mem-sub memory operations in the E-Set at  
diamond 614, the "NO" exit is taken out of diamond 614 and block 618 entered.  
The action taken in block 618 is to choose a next best node from the ready set,  
20 given the fact that there are no memory subtract node types available. The same  
types of heuristics discussed when box 618 was entered earlier apply here, as are  
well known in the art. After a node is chosen using a heuristic, the actions taken

in block 618 are, first, removing the chosen node from the ready set. The element in the memory queue corresponding to node just removed from the ready set (if there is one), is removed from the memory queue along with any elements closer to the end of the queue (if any), in keeping with FIFO ordering. Next, the 5 instruction corresponding to the chosen node is inserted in the instruction stream. Finally, the chosen node is removed from the DAG, including the removal of any dependency edges originating from that node.

After these actions are completed, block 618 is exited and block 602 10 entered. The process will now continue with block 602, carrying out the DAG traversal and identification of the current ready set.

The method for the present invention has been described primarily in terms 15 of using a single memory queue to model the actual memory queue or queues in a target processor. However, it is fully contemplated that the present invention encompasses any number of queues that may be needed to most efficiently model the behavior of any or all target processors. "Target processor(s)" is being used to mean the processor or processor family on which the compiled code is designed to run. This may or may not be the same as the processor on which the system 20 embodying the present is installed. For many applications the two will definitely be different (i.e., embedded systems development).

In implementations where multiple memory queues are being used rather than a memop single queue, a few modifications to the description of the logical blocks need to be made. In FIG. 5 changes will appear primarily in ready set evaluation module 512 and in the queue management module 514. Queue management module 514 can readily handle any number of independent queues, each with a unique name. One possible implementation would have three queues: a load queue (e.g., fed by load instructions and drained by register usage); a prefetch queue (e.g., fed by prefetch instructions and drained by loads and stores); and, a store queue (e.g., for store instructions). Ready set evaluation module 512 would know about each queue, and would handle memory operations that add and remove memory operations in a way that is in accordance with the effects on the target processor(s). For example, a priority scheme where the prefetch queue has highest priority and the store queue has the lowest priority is part of the implementation. Thus, the present invention fully contemplates the use of one or more memory operation queues (normally expected to be 1 to 3 but expandable as needed), where the number of queues used will depend on the best way of modeling the target processor(s). This also includes implementations where the number of queues in use may be dynamic, that is, dependent on both the target processor(s) and the level of instruction scheduling that is desirable for the program being compiled.

The description of the current invention used, for illustrative purposes, a FIFO queue having a maximum queue size (or number of elements) defined as (queue\_size - 1). Queue\_size is determined by looking at the target processor. This was to help with intuitive understanding of the relationship between the 5 queue used in the present invention and an actual hardware queue found in a target processor. As will be obvious to one of ordinary skill in the art and with the benefit of the present disclosure, other queue disciplines coupled with an indicator for a desired number of outstanding memory operations may be used in the present invention while remaining entirely within the inventive bounds of the present 10 disclosure. Such variations are fully contemplated by the current disclosure.

The present invention has been partially described through the use of a flow chart. As is appreciated by those of ordinary skill in the art and with the benefit of the present disclosure, the procedures described herein may be repeated as 15 continuously, as often, or as little as necessary to satisfy the needs described and details or order of steps may vary without departing from the basic concepts of the present invention.

A system embodying the present invention could run on any computer 20 system having a standard configuration as a development system; it can also be the case that some components of the present invention could be implemented in hardware such as in a hardware scheduler, and such a system would then include

those portions in software and hardware. The software portions of the present invention is embodied in various machine readable media on the system, on which are stored programs, data, and data structures. It is contemplated that any media suitable for use with the program and its accompanying data and data structures is 5 within the scope of the present invention. Examples would include memory and other relatively fast-access semiconductor media, as well as magnetic or optical secondary storage media.

10 While embodiments and applications of this invention have been shown and described, it will be apparent to those of ordinary skill in the art and with the benefit of the present disclosure that many more modifications than mentioned above are possible without departing from the inventive concepts contained herein. The invention, therefore, is not to be restricted except in the spirit of the associated claims.